High Performance Beowulf Cluster Environment

User Manual

Version 1.8

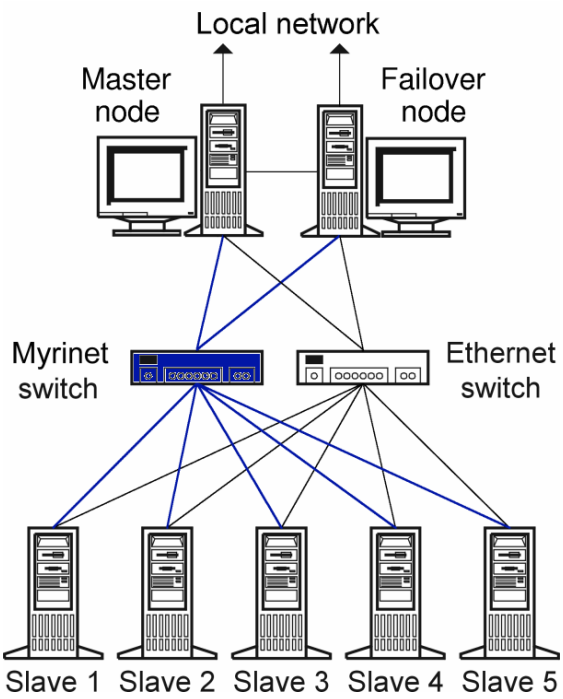
Introduction

This guide is intended for cluster users who need a quick introduction to the ClusterVision Beowulf Cluster Environment. It explains how to use the MPI and batch environments, how to submit jobs to the queuing system, and how to check job progress.

The specific combination of hardware and software installed may differ depending on the specification of the cluster. This manual may refer to hardware, libraries or compilers not relevant to the environment at hand.

The physical hardware layout of a Cluster

A Beowulf Cluster consists of a login, compile and job submission node, called the master [or management] node, and one or more compute nodes, normally referred to as slave [or worker] nodes. A second (fail-over) master may be present in order to take control of the cluster in case the main master node fails. Furthermore, a second fast network may also have been installed for high performance communication between the (master and the) slave nodes (see figure).



The master node is used to compile software, to submit a parallel or batch program to a job queuing system and to gather/analyse results. Therefore, it should rarely be necessary for a user to log on to one of the slave nodes and in some cases slave node logins are disabled altogether.

The master and slave nodes communicate with each other through an *Ethernet* network. Depending on the situation either a Fast Ethernet or a Gigabit Ethernet is in use, the former capable of transmitting information at a maximum rate of 100 Megabits/s and the latter at a maximum rate of 1000 Megabits/s.

Sometimes an additional network is added to the cluster for even faster communication between [the master and] the slave nodes. This particular network is mainly used for programs dedicated to solving large scale computational problems, which may require multiple machines and could involve the exchange of vast amounts of information.

One such network topology is *Myrinet*, capable of transmitting information at a maximum rate of 4000 Megabits/s. Another cluster interconnect network which could be in use is *Infiniband,* available in a variety of transmission speeds e.g. 2.5 Gigabit/s or 4 x 2.5 Gigabit/s. When installed, these networks are always complementary to a slower Ethernet based network.

Cluster nodes may be specified with 32 bit Intel or AMD CPUs. Alternatively, 64bit AMD Opteron CPUs may be specified. This is referred to as the 'x86-64' architecture. The advantage of the 64bit architecture lies in its capability to natively address memory beyond the 4 Gigabyte limitation of 32-bit processors. Hardware can thus be simplified to utilize far more on-board memory [RAM], essential for such large scale computational tasks as mentioned above.

Accessing the cluster

The master node is usually assigned a name and an IP address by the local network administrator. Outside the cluster it is usually referred to by this name only. Within the cluster the master is also accessible by the alias `master`. The slave nodes can only be accessed from within the cluster (i.e. by logging on to the master first) by the aliases `node01`, `node02`, etc.

For access to the master node from the local network the use of `ssh` (Secure SHell) is highly recommended. The command structure for ssh is as follows:

`ssh [-l login_name] hostname | user@hostname [command]`.

So for the user '`john`' and the master node '`cluster.org`' the command `ssh john@cluster.org` will log you on to the cluster.

Once logged on to the master node, access to the slave nodes is obtained either by log on or by remotely executing commands. To log on to a slave node, use `ssh` or `rlogin`. To execute a command, use `rsh`. (The ClusterVision environment has special utilities to execute commands in parallel on all or only a group of nodes).

Discovering your cluster

Before a parallel or batch program can be run it is necessary to know which directories are available for storing data, which queues are available for executing programs, and how many slave nodes are part of the queuing system. Furthermore the account(s) used for this purpose must have sufficient access to run programs on the slave nodes.

The commands listed in the following table should indicate whether *Gridengine* is in use as a queuing system:

Command	Action
`qconf –sql`	shows available queues
`qhost`	shows the slave nodes that are part of the queuing system and their status
`qconf –spl`	shows all available parallel environment
`qhost –q`	shows the slave nodes and the queues associated with each
`pexec date`	Executes the program 'date' on all the slave nodes. This requires permission to rsh to a slave node.

The following three commands should indicate whether *PBS* is the queuing system in use:

Command	Action
`qstat –Q`	shows the available queues
`pbsnodes -a`	shows the slave nodes that are part of the queuing system and their status
`pexec date`	Executes the program 'date' on all the slave nodes. This requires permission to rsh to a slave node.

If all the above commands fail, please check the section on Environment Modules below. It is more than likely a correct environment module needs to be loaded/installed first.

Important directories to be found on the master and slave nodes:

`/home/<user_name>/`	User home directory, which is NFS-mounted from the master node.
`/usr/local/`	Program and library directories, which is NFS-mounted from the master.
`/usr/local/Cluster-Examples/`	Directory with examples to help you get started.
`/usr/local/Cluster-Docs/`	Directory with all cluster documents
`/data, /scratch or similar`	Local scratch space. The usage of this directory depends on the site policy. In some cases, you first need to ask your administrator for a subdirectory.

The Modules Environment

On a complex computer system with (often) a (wide) choice of software packages and software versions it can be quite hard to set up the correct environment to manage this. For instance, managing diverse MPI software packages on the same system or even different versions of the same MPI software package is almost impossible on a standard SuSE or Red Hat system as many software packages use the same names for executables and libraries.

As a user you could end up with the problem that you could never be quite sure which libraries have been used for the compilation of a program as multiple libraries with the same name may be installed. Very often a user would like to test new versions of a software package before permanently installing the package. Within Red Hat or SuSE this would be quite a complex task to achieve. The `module` environment makes this process easy.

The command to use is `module`:

`module`

`(no arguments)`	print usage instructions
`avail`	list available software modules
`load modulename`	add a module to your environment
`add modulename`	add a module to your environment
`unload modulename`	remove a module
`clear`	remove all modules

This is an example output from `module avail`:

```
------------------------/usr/local/Cluster-Config/Modulesfiles/------------
acrobat/5.0.8          gromacs/3.2-mpich-gm          nagios/1.1
bonnie++/1.03a         intel-compiler/7.1            null
cluster-tools/0.9      intel-compiler/8.0            pbs/1.0.1
cluster-tools/1.0      intel-math/6.1                pgi/5.1-2
clusteradmin/0.1.1     module-info                   pgi/5.1-3
default-ethernet       modules                              sge/5.3p5
default-myrinet        mpich/1.2.5.2-1-gnu323               superodoctor/1.2.016
dot                    mpich/1.2.5.2-1-intel71       totalview/6.3.0-1
fftw/2.1.3             mpich-gm/1.2.5..10-gnu323     use.own
gm/2.0.8               mpich-gm/1.2.5..10-intel71    version
gromacs/3.2-mpich      mute/1.9.5
```

This indicates that there are several different modules available on the system. For example, `cluster-tools` is the Clustervision management and control suite. It is also possible to make use of various compilers and parallel libraries, to be described in later sections.
The computer administrator should make a default module available for use by logged-in users and jobs running on the system. In the example above this could be the `default-ethernet` module.
The module `default-ethernet` loads several other modules, namely the Intel compiler, the Intel math libraries, `cluster-tools`, sge (the queue manager) and `mpich` for Ethernet. If a fast network is available e.g. Myrinet it would preferable to load the `default-myrinet` module.

In order to ensure that the environment remains the same at login time, it is best to load the modules from the `.bashrc`, `.tcshrc` or `.cshrc` script files. These files can be created from scratch/manually or are already located in the home directory and are executed every time a master node login takes place. If the Bourne-Again Shell (`bash`)is in use, the `.bashrc` file needs to be updated with the proper commands. If otherwise the `csh` or `tcsh` shell are in use, the `.cshrc` or `.tcshrc` files respectively need updating. Since the home directory is shared with the slave nodes, the same scripts are also executed when a user's job commences on one of the slave nodes.

It is important to choose the environment that reflects the job(s) which need running. For batch applications (i.e. applications that do not involve more than one cpu), a different environment can be defined in the script submitted to the queuing system. For parallel applications this is possible too, however only the first MPI thread will inherit this environment.

Creating your own modules (Advanced users)

It is possible to create new modules and add them to the module environment. After installing an application in (for example) `~/my_application` and creating a new module directory in `~/my_modules`, the `MODULEPATH` can be modified to include the new search path. If you use the bash shell this command would be:

```
export MODULEPATH=$MODULEPATH:~/my_modules
```

To make this change permanent, just add this command to your `.bashrc`, `.tcshrc` or `.cshrc` file. The contents of a module look like this:

```
##%Module1.0#####################################
##
## mpich-gnu modulefile
##
proc ModulesHelp { } {

        puts stderr "\tAdds myrinet mute to your environment"
}

module-whatis   "Adds myrinet mute to your environment"

set            root                   usr/local/Cluster-Apps/mute-1.9.5
setenv         GM_HOME                       $root
setenv         gm_home                       $root
append-path    PATH                   $root/bin/
append-path    MANPATH                       $root/man
append-path    LD_LIBRARY_PATH        $root/lib/
```

Typically one would only have to fill in the root path (`/usr/local/Cluster-Apps/mute-1.9.5`) and the description to make a new and fully functional module. For more information, please load the 'modules' module (`module load modules`), and read the *module* and *modulefile* man-pages. The best policy is to create an additional directory within the modules directory for each separate application and to place the new module in that directory. Then the name of the module file can be altered so that it reflects the version number of the application. For instance this is the name of the location of the `mute` module: `/usr/local/Cluster-Config/Modulefiles/mute/1.9.5` Now by issuing the command '`module load mute`' the new module will automatically load. The advantage to this is that if different versions of the same application are installed, this command would always load the most recent version.

Running your application on the cluster

In order to use and run an application on the cluster, four steps are necessary:

- selecting the parallel environment
- compiling the application
- creating a job script for the queuing system
- submitting the application to the queuing system

Step 1: Selecting your parallel environment

In order to run parallel code, a specific parallel set up is necessitated. Here the choices are between the various types of network, MPI libraries and compilers.

There may be different types of MPI libraries on the cluster: MPICH for Ethernet based communication, MPICH-GM for Myrinet and MPICH-VAPI for Infiniband.

There may also be a choice of compilers: the open source GNU gcc, or the higher performance compilers from Portland or Intel.

For selecting between the various compilers and MPI libraries, use the `module` environment. During a single login session, just execute the command-line tool `module`. This facilitates switching between software versions and will set variables to the executables e.g. `$PATH`. However, such changes will only be in effect for the present login session, and subsequent sessions will not use these settings.

To make a combination of compiler and MPI library permanent, edit the `.bashrc` file (assuming the bash shell), inserting a line as follows:

```
module add mpich/1252-intel
```

The naming scheme for this combination is as follows: the first part of the name refers to the type of MPI library: this is `mpich` for MPI over normal ethernet, '`mpich_gm`' for MPI over Myrinet and '`mvapich`' for MPI over Infiniband. This is followed by the version number of the MPI library. The last part of the name is that of the compiler to be used when the `mpicc` command is run.

Step 2: Compiling your code:

Typically, there are several compilers available on the master node. A few examples: GNU, Intel and Portland Group compilers. The following table summarises the available compiler commands on the cluster:

Language	GNU compiler	Portland compiler	Intel compiler
C	gcc	pgcc	icc
C++	c++	pgCC	icc
Fortran77	f77	pgf77	Ifc (ifort for v8.0)
Fortran90	-	pgf90	Ifc (ifort for v8.0)

The most common code optimisation flag for the GNU compiler is `-03` and for the Portland compiler `-fast`. There is no GNU compiler for Fortran90. For maximum application speed, use of Portland and/or Intel compilers is recommended.
Please refer to the respective man-pages for more information about optimisation for both GNU and Portland.HTML and PDF documentation for Portland may be found in the `/usr/local/Cluster-Docs` directory .

The commands referred to in the table are specific for batch type (single processor) applications. For parallel applications it is preferable to use MPI based compilers. The necessary compilers are automatically available after choosing the parallel environment. The following compiler commands are available:

Code	Compiler:
C	mpicc
C++	mpiCC
Fortran77	mpif77
Fortran90	mpif90

These MPI compilers are ‘wrappers’ around the GNU, Portland and Intel compilers and ensure that the correct MPI include and library files are linked into the application (Dynamic MPI libraries are not available on the cluster). Since they are wrappers, the same optimisation flags can be used as with the standard GNU or Portland compilers.

Typically, applications use a Makefile that has to be adapted for compilation. Please refer to the application’s documentation in order to adapt the Makefile for a Beowulf cluster. Frequently, it is sufficient to choose a Makefile specifically for a Linux MPI environment and to adapt the `FC` and `FF` parameters in the Makefile. These parameters should point to `mpicc` and `mpif77` (or `mpif90` in the case of Fortran90 code) respectively.

Step 3: Executing the program

There are two methods for executing a parallel or batch program: using the queuing system or directly from the command line. In general it is preferable to use the queuing system, particularly in a production environment with multiple users. On some systems, running parallel programs outside the queuing system may be disabled. However, if a quick test of the application is necessary you may be able to use the command line and run it outside the queuing system.

Non-parallel programs can most easily be run straight from the slave nodes. This can be achieved by logging into one of the slave nodes using `rlogin` or `rsh`, and changing to the directory where your application resides and execute it. It is also possible to execute a program remotely on any node by typing:

`rsh <node name> <program>`.

For example, to run the `date` command on node02 type:
`rsh node02 date.`

Refer to the `rsh` man page for further details. Please note that on some systems, where running jobs outside the queuing system is not allowed, this will not work.

Running a parallel program is slightly more complicated. All installed parallel MPI environments need to know on which slave nodes to run the program. The methods for telling the program which nodes to use differ however.

Using MPICH

The command line for MPICH (TCP/IP based communication) would look like this:

```
mpirun -machinefile configuration_file -np 4 program_name program_options
```

The configuration file would look like this:

```
node02
node02
node03
node03
```

Which is equivalent to:

```
node02:2
node03:2
```

The `:2` extension informs MPICH that two processes are to be run on each node. Please refer to the specific man-pages and the command `mpirun -h` for more information.

Using MPICH-VAPI

If the cluster is equipped with InfiniBand networking, the MPICH-VAPI package should be used. MPICH-VAPI is actually a patched version of MVICH. MVICH is an implementation of MPICH which accesses Ethernet hardware directly, instead of going through the TCP/IP stack. Accordingly, MPICH-VAPI accesses the InfiniBand networking hardware directly, thus eliminating TCP/IP protocol overhead.

The `mpirun` differs a bit from the standard MPICH `mpirun` command. Actually it is an `mpirun_rsh` wrapper.

The command line for `mpirun` would look like this:

```
mpirun -hostfile host_configuration_file –np number_of_processes  program_name
program_options
```

The configuration file would look like this:

```
node02
node02
node03
node03
node03
node04
```

Which is equivalent to:

```
node02:2
node03:3
node04:1
```

This is a host file for 6 processes. Two are started on node2, three on node3 and a single process on node4. Of course, this is just an example. On dual processor clusters, you would probably want to run only two processes per node.

Note: if multiple processes are to be run on one machine, it is preferable that all lines referring to this machine be listed consecutively.

Please refer to the MVICH documentation and the command `mpirun_rsh -h` for more information.

Using MPICH-GM

A typical command line for MPICH-GM (Myrinet based communication) in the directory in which the program is located, is the following:

```
mpirun.ch_gm  --gm-kill  1  --gm-f  configuration_file  -np  4  program_name
program_options
```

The configuration_file consists typically of the following lines:

```
node02
node02
node03
node03
```

Which is equivalent to:

```
node02:2
node03:2
```

The configuration_file example shows that the application using this configuration file will be started with two processes on each node, as in the case of dual CPU slave nodes.

The -np switch on the mpirun.ch_gm command line indicates the number of processes to run, in this case 4.

The total list of options is:

```
mpirun.ch_gm [--gm-v] [-np <n>] [--gm-f <file>] [--gm-h] prog [options]
```

Option	Explanation
--gm-v	verbose - includes comments
-np <n>	specifies the number of processes to run
--gm-np <n>	same as -np (use one or the other)
--gm-f <file>	specifies a configuration file
--gm-use-shmem	enable the shared memory support
--gm-shmem-file <file>	specifies a shared memory file name
--gm-shf	explicitly removes the shared memory file
--gm-h	generates this (help) message
--gm-r	start machines in reverse order
--gm-w <n>	wait n secs between starting each machine [job?]
--gm-kill <n>	n secs after first process exits, kill all other processes
--gm-dryrun	Don't actually execute the commands, just print them
--gm-recv <mode>	specifies the recv mode, 'polling', 'blocking' or 'hybrid'
--gm-recv-verb	specifies verbose for recv mode selection
-tv	specifies totalview debugger

Use of the option --gm-use-shmem is highly recommended as it improves performance. Another recommended option is --gm-kill. It is possible for a program to get 'stuck' on a machine and MPI is unable to pick up the error. The problem is that the program keeps the port locked and no other program will be able to use the Myrinet port. The only way to fix this is to manually kill the MPI program on the slave node. If the option

`--gm-kill 1` is used, MPI makes a better effort to properly kill programs after a failure.

An example MPI program

The sample code below contains the complete communications skeleton for a dynamically load balanced master/slave application. Following the code is a description of some of the functions necessary for writing typical parallel applications.

```
#include <mpi.h>
#define WORKTAG     1
#define DIETAG     2
main(argc, argv)
int argc;
char *argv[];
{
       int         myrank;
       MPI_Init(&argc, &argv);   /* initialize MPI */
       MPI_Comm_rank(
       MPI_COMM_WORLD,   /* always use this */
       &myrank);      /* process rank, 0 thru N-1 */
       if (myrank == 0) {
              master();
       } else {
              slave();
       }
       MPI_Finalize();       /* cleanup MPI */
}

master()
{
       int    ntasks, rank, work;
       double       result;
       MPI_Status     status;
       MPI_Comm_size(
       MPI_COMM_WORLD,   /* always use this */
       &ntasks);          /* #processes in application */
/*
* Seed the slaves.
*/
       for (rank = 1; rank < ntasks; ++rank) {
              work = /* get_next_work_request */;
              MPI_Send(&work,         /* message buffer */
              1,              /* one data item */
              MPI_INT,        /* data item is an integer */
              rank,           /* destination process rank */
              WORKTAG,        /* user chosen message tag */
              MPI_COMM_WORLD);/* always use this */
       }

/*
* Receive a result from any slave and dispatch a new work
* request work requests have been exhausted.
*/
       work = /* get_next_work_request */;
       while (/* valid new work request */) {
              MPI_Recv(&result,       /* message buffer */
              1,              /* one data item */
              MPI_DOUBLE,     /* of type double real */
              MPI_ANY_SOURCE, /* receive from any sender */
              MPI_ANY_TAG,    /* any type of message */
              MPI_COMM_WORLD, /* always use this */
              &status);       /* received message info */
              MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
```

```
                WORKTAG, MPI_COMM_WORLD);
                work = /* get_next_work_request */;
        }
/*
* Receive results for outstanding work requests.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
/*
* Tell all the slaves to exit.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
        }
}

slave()
{
        double              result;
        int                 work;
        MPI_Status          status;
        for (;;) {
                MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
/*
* Check the tag of the received message.
*/
                if (status.MPI_TAG == DIETAG) {
                        return;
                }
                result = /* do the work */;
                MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
}
```

Processes are represented by a unique rank (integer) and ranks are numbered 0, 1, 2, ..., N-1. `MPI_COMM_WORLD` means all the processes in the MPI application. It is called a communicator and it provides all information necessary to do message passing. Portable libraries do more with communicators to provide synchronisation protection that most other systems cannot handle.

Enter and Exit MPI

As with other systems, two functions are provided to initialise and clean up an MPI process:

```
        MPI_Init(&argc, &argv);
        MPI_Finalize( );
```

Who Am I? Who Are They?

Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist. A process finds out its own rank by calling:

```
MPI_Comm_rank( ):
   Int myrank;
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
The total number of processes is returned by MPI_Comm_size( ):
   int nprocs;
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Sending messages

A message is an array of elements of a given data type. MPI supports all the basic data types and allows a more elaborate application to construct new data types at runtime. A message is sent to a specific process and is marked by a tag (integer value) specified by the user. Tags are used to distinguish between different message types a process might send/receive. In the sample code above, the tag is used to distinguish between work and termination messages.

```
MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD);
```

Receiving messages

A receiving process specifies the tag and the rank of the sending process. `MPI_ANY_TAG` and `MPI_ANY_SOURCE` may be used optionally to receive a message of any tag and from any sending process.

```
MPI_Recv(buffer, maxcount, datatype, source, tag,
MPI_COMM_WORLD, &status);
```

Information about the received message is returned in a status variable. The received message tag is status. `MPI_TAG` and the rank of the sending process is `status.MPI_SOURCE`.

Another function, not used in the sample code, returns the number of data type elements received. It is used when the number of elements received might be smaller than `maxcount`.

```
        MPI_Get_count(&status, datatype, &nelements);
```

With these few functions, you are ready to program almost any application. There are many other, more exotic functions in MPI, but all can be built upon those presented here so far.

Step 4: Running your program from the queuing system

A workload management system – also known as a batch or queuing system – allows users to make more efficient use of their time by allowing you to specify tasks to run on the cluster. It allows administrators to make more efficient use of cluster resources, spreading load across processors and scheduling tasks according to resource needs and priority.

The queuing system allows parallel and batch programs to be executed on the cluster. The user asks the queuing system for resources and the queuing system will reserve machines for execution of the program.

The user submits jobs and instructions to the queuing system through a shell script. The script will be executed by the queuing system on one machine only; the commands in the script will have to make sure that it starts the actual application on the machines the queuing system has assigned for the job. The system takes care of holding jobs until computing resources are available, scheduling and executing jobs and returning the results to you.

The cluster will use either the Portable Batch System or Gridengine.

Portable Batch System

The Portable Batch System (PBS) is a workload management and job scheduling system first developed to manage computing resources at NASA.

PBSPro is the professional version of the Portable Batch System `http://www.pbspro.com`

Information about OpenPBS, an older open source release can be found at `http://www.openpbs.org`

PBS has both a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs. To use PBS, you create a *batch job* which is a shell script containing the set of commands you want to run. It also contains the resource requirements for the job. The batch job script is then submitted to PBS. A job script can be resubmitted with different parameters (e.g. different sets of data or variables).

Sample batch submission script

This is a small example script, used to submit non-parallel jobs to PBS.

```
#!/bin/bash
#
#PBS -l walltime=1:00:00
#PBS -l mem=500mb
#PBS -j oe

cd ${HOME}/myprogs
myprog a b c
```

The lines beginning #PBS are directives to the batch system. The directives with `-l` are resource directives, which specify arguments to the `-l` option of qsub. In this case, a job time of one hour and at least 500Mb are requested. The directive `-j oe` requests standard out and standard error to be combined in the same file. PBS stops reading directives at the first blank line. The last two lines simply say to change to the directory myprogs and then run the executable `myprog` with arguments `a b c`.

Sample PBS script for MPICH-GM

A more complicated PBS script for Myrinet based MPICH-GM looks like this:

```
#!/bin/csh
#
# Tell PBS to use 2 nodes and 1 process per node
#PBS -l nodes=2:ppn=1
```

```
# Tell PBS to reserve a maximum of 12 hours and 15 minutes
#PBS -l walltime=12:15:00

#################################################################
############### Config script for using MPICH-GM #############
#################################################################

# Set the following entries:

# Run dir:
set RUNDIR = "/home/mpi"

# Application name:
set APPLICATION = "PMB-MPI1"

# Extra flags for Application
set RUNFLAGS = ""

# Extra flags for mpich:
set EXTRAMPI = ""


#################################################################
#        Below this nothing should have to be changed        #
#################################################################

echo Running from MPI $MPI_HOME
echo
echo Changing to $RUNDIR
cd $RUNDIR

set nodes = `cat $PBS_NODEFILE`
echo $nodes > /tmp/$PBS_JOBID.conf
set nnodes = $#nodes

set confile = /tmp/$PBS_JOBID.conf

cat $confile

echo "Will run command: mpirun.ch_gm -np $nnodes -machinefile $confile
$EXTRAMPI $APPLICATION $RUNFLAGS"
echo Starting job...
time mpirun.ch_gm -np $nnodes -machinefile $confile $EXTRAMPI $APPLICATION
$RUNFLAGS
rm -rf $confile
```

As can be seen in the script, a configuration file for Myrinet is built using the `$PBS_NODEFILE` variable. This variable is supplied by the queuing system and contains the node names that are reserved by the queuing system for running the job. The configuration file is given a unique name (`/tmp/$PBS_JOBID.conf`) in order to make sure that users can run multiple programs concurrently.

Submitting the job to the queuing system

The command `qsub` is used to submit jobs. The command will return a unique job identifier, which is used to query and control the job and to identify output. See the respective man-page for more options.

`qsub scriptname`	submits a script for execution
`-a datetime`	run the job at a certain time
`-l list`	request certain resource(s)

-q	queue	jobs is run in this queue
-N	name	name of job
-S	shell	shell to run job under
-j	oe	join output and error files

Submitting the script to the queuing system

The command `qstat -an` shows what jobs are currently submitted in the queuing system and the command `qstat -q` shows what queues are available An example output is:

```
qstat -an:
hpc.ClusterVision.co.uk:
                                                            Req'd  Req'd   Elap
Job ID      Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
----------- -------- -------- ---------- ------ --- --- ------ ----- - -----
394.master. mpi      long     pbs_submit   5525  16  --    --  12:00 R   --
   node17/1+node17/0+node16/1+node16/0+node15/1+node15/0+node14/1+node14/0
   +node13/1+node13/0+node12/1+node12/0+node11/1+node11/0+node10/1+node10/0
   +node9/1+node9/0+node8/1+node8/0+node7/1+node7/0+node6/1+node6/0+node5/1
   +node5/0+node4/1+node4/0+node3/1+node3/0+node2/1+node2/0

qstat -q:
server: master.beo.org

Queue            Memory CPU Time Walltime Node Run Que Lm  State
---------------- ------ -------- -------- ---- --- --- --  -----
long               --      --    12:00:00  --    0   0 10   E R
default            --      --       --     --    0   0 10   E R
small              --      --    00:20:00  --    0   0 10   E R
verylong           --      --    72:00:00  --    0   0 10   E R
medium             --      --    02:00:00  --    0   0 10   E R
                                               --- ---
                                                 0   0
```

The `qstat -q` command in this case shows that there are 5 queues: `long`, `default`, `small`, `verylong` and `medium`. The default queue is a so-called ‘routing queue’ and routes jobs to other queues depending on the needed resources. The `Time` entry in the table shows the maximum time a job may be running in a queue.

It is recommended that you always use the `#PBS -l walltime=00:00:00` directive. This allows the queuing system to automatically choose the right queue for you and it makes the scheduler more efficient.

Gridengine

Gridengine is a package of software for distributed resource management on compute clusters and grids. Utilities are available for job submission, queuing, monitoring and checkpointing. The Gridengine software is made available by Sun Microsystems and the project homepage is at `http://gridengine.sunsource.net`, where extensive documentation can be found.

Sample batch submission script

This is an example script, used to submit non-parallel scripts to Gridengine. This script can be found in `/usr/local/Cluster-Docs/examples`.

```
#!/bin/bash
#
# Script to submit a batch job
#
# ---------------------------
# Replace these with the name of the executable
# and the parameters it needs
# ---------------------------
export MYAPP=/home/myname/codes/mycode
export MYAPP_FLAGS='1 2 3'

# ---------------------------
# set the name of the job
#$ -N example_job

######################################################################
######################################################################
# there shouldn't be a need to change anything below this line

#---------------------------
# set up the parameters for qsub
# ---------------------------

#  Mail to user at beginning/end/abort/on suspension
#$ -m beas
#  By default, mail is sent to the submitting user
#  Use  $ -M username    to direct mail to another userid

# Execute the job from the current working directory
# Job output will appear in this directory
#$ -cwd
#   can use -o dirname to redirect stdout
#   can use -e dirname to redirect stderr

# for submission to express queue,
# either use -l express on the command line i
# or use #$ -l express in this script
#$ -l express

#  Export these environment variables
#$ -v PATH

export PATH=$TMPDIR:$PATH

# ---------------------------
# run the job
# ---------------------------
```

```
$MYAPP $MYAPP_FLAGS
```

In order to submit a script to the queuing system, the user issues the command `qsub scriptname`. Usage of `qsub` and other queueing system commands will be discussed in the next section.

Note that `qsub` accepts only shell scripts, not executable files. All options accepted by the command-line `qsub` can be embedded in the script using lines starting with `#$` (see below).

The script above first sets the name of the executable file which will be run by the batch job. Substitute the path and name of your own executable. It next sets any input parameters which the code might expect. For example, if you run the code on the command line using `mycode x 20 y 20` set `MYAPP_FLAGS` to `x 20 y 20`. The name of the job is set to `example_job`: job output `.o` and error `.e` files will use this name, appended with the job-id.

The `-m` flag is used to request the user be emailed. The job output is directed into the directory you are currently working from by `-cwd`. The sample script requests the resource `express` by using the `-l` flag.

When a job is submitted, the queuing system takes into account the requested resources and allocates the job to a queue which can satisfy these resources. If more than one queue is available, load criteria determine which node is used, so there is load balancing. There are several requestable resources, including system memory and max CPU time for a job.

Sample script for MPICH

This is an example script, used to submit MPICH jobs to Gridengine.

```
#!/bin/bash
#
# Script to submit an mpi job

# ----------------------------
# Replace these with the name of the executable
# and the parameters it needs
export MYAPP=/home/myname/codes/mycode
export MYAPP_FLAGS='1 2 3'

# ----------------------------
# set the name of the job
#$ -N example_job

# ----------------------------
# select the MPICH version you want to use
export MYMPIVER='mpich1252/gcc-3.3-64b'

######################################################################
```

```
#########################################################################
# there shouldn't be a need to change anything below this line




# ----------------------------
# set up the mpich version to use
# ----------------------------


# load the module
if [ -f /etc/profile.modules ]
then
        . /etc/profile.modules
        module load null $MYMPIVER
fi

#----------------------------
# set up the parameters for qsub
# ----------------------------

#  Mail to user at beginning/end/abort/on suspension
#$ -m beas
#  By default, mail is sent to the submitting user
#  Use  $ -M username    to direct mail to another userid

# Execute the job from the current working directory
# Job output will appear in this directory
#$ -cwd
#   can use -o dirname to redirect stdout
#   can use -e dirname to redirect stderr

#  Export these environment variables
#$ -v PATH
#$ -v MPI_HOME

# select the parallel enviroment
# and request between 2 and 8 slots
#$ -pe mpi 2-8

# Gridengine allocates the max number of free slots and sets the
# variable $NSLOTS.
echo "Got $NSLOTS slots."

# Gridengine sets also $TMPDIR and writes to $TMPDIR/machines the
# corresponding list of nodes. It also generates some special scripts in
# $TMPDIR. Therefore, the next two lines are practically canonical:
#
export PATH=$TMPDIR:$PATH

# ----------------------------
# run the job
# ----------------------------
echo   "Will   run   command:   $MPI_HOME/bin/mpirun   -np   $NSLOTS   -machinefile
$TMPDIR/machines  $MYAPP $MYAPP_FLAGS"

$MPI_HOME/bin/mpirun   -np   $NSLOTS   -machinefile   $TMPDIR/machines     $MYAPP
$MYAPP_FLAGS
```

Example scripts can be found in `/usr/local/Cluster-Docs/examples/`. The script sets the name of the executable, any parameters needed by the user code and a name for the job. It requests from 2 up to 8 parallel slots and runs the job.

Gridengine queuing

The Gridengine queuing system is used to submit and control batch jobs. The queuing system takes care of allocating resources, accounting and load balancing jobs across the cluster. Job output and error messages are returned to the user. The most convenient method to update or see what happens with the queuing system is using the (with Motif compiled) `qmon`.

The command `qstat` is used to show the status of queues and submitted jobs.

`qstat`	show job/queue status
`no arguments`	show currently running/pending jobs
`-f`	show full listing of all queues, their status and jobs
`-j [job_list]`	shows detailed information on pending/running job
`-u user`	shows current jobs by user

The command `qhost` is used to show information about nodes.

`qhost`	show job/host status
`no arguments`	show a table of all execution hosts and information about their configuration
`-l attr=val`	show only certain hosts
`-j [job_list]`	shows detailed information on pending/running job
`-q`	shows detailed information on queues at each host

The command `qsub` is used to submit jobs.

`qsub scriptname`	submits a script for execution
`-a time`	run the job at a certain time
`-l`	request a certain resource
`-q queues`	jobs is run in one of these queues

There are many other options to qsub, consult the man page for more information.

The command `qdel` is used to delete jobs.

`qdel jobid(s)`	deletes one (or more) jobs
`-u username`	deletes all jobs for that user
`-f`	force deletion of running jobs

The command `qmod` is used by administrators to modify, suspend and restart queues.

`qmod queue_list`	modifies a queue or queues

`-c`	clears a reported error state
`-d`	disables the queue
`-e`	enables the queue
`-r`	reschedules the jobs
`-s`	suspends the queue and jobs
`-us`	unsuspends the queue and jobs